NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TECHNIQUES
FOR
MULTIPLE DATABASE INTEGRATION

by

Barron D. Whitaker

March 1997

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE TECHNIQUES FOR MULTIPLE DATABASE INTEGRATION	5. FUNDING NUMBERS
6. AUTHOR(S) Whitaker, Barron D.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

There are several graphic client/server application development tools which can be used to easily develop powerful relational database applications. However, they do not provide a direct means of performing queries which require relational joins across multiple database boundaries.

This thesis studies ways to access multiple databases. Specifically, it examines how a "cross-database join" can be performed. A case study of techniques used to perform joins between academic department financial management system and course management system databases was done using PowerBuilder 5.0.

Although we were able to perform joins across database boundaries, we found that PowerBuilder is not conducive to cross-database join access because no relational database engine is available to execute cross-database queries.

14. SUBJECT TERMS Relational Database, Object Based Database Systems Design, Multiple Database Integration	15. NUMBER OF PAGES 77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITAITON OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18-298-102

TECHNIQUES FOR MULTIPLE DATABASE INTEGRATION

Barron D.Whitaker
Major, United States Marine Corps
B.S., University of Texas A & M, 1984

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1997

Author: [illegible]
Barron D. Whitaker

Approved by: [illegible]
C. Thomas Wu, Thesis Advisor

[illegible]
Mark H. Barber, Second Reader

[illegible]
Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

There are several graphic client/server application development tools which can be used to easily develop powerful relational database applications. However, they do not provide a direct means of performing queries which require relational joins across multiple database boundaries.

This thesis studies ways to access multiple databases. Specifically, it examines how a “cross-database join” can be performed. A case study of techniques used to perform joins between academic department financial management system and course management system databases was done using PowerBuilder 5.0.

Although we were able to perform joins across database boundaries, we found that PowerBuilder is not conducive to cross-database join access because no relational database engine is available to execute cross-database queries.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

Organizations use database systems and applications to process information needed to conduct daily business and operations. Historically, these database systems were individually developed to meet specific needs. As an organization's information requirements changed over time, additional applications were written to maintain, process, and extract information from the individual databases. Typically however, each of these databases provided only a part of the total picture needed by an organization. An example of this is illustrated in Figure 1. In this example integration of the two databases makes it possible to perform a query to retrieve the answer to the question, "How many sophomores have books checked out?". However, this information is not available from either of the preexisting underlying databases in this example. Additionally, as a result of the way that databases in organizations tend to evolve, it often becomes necessary to maintain databases that contain data which is duplicated in other databases. This leads to the additional problem of how to deal with data inconsistencies between the differing systems. This situation is inefficient both from a management perspective and from a data processing perspective since it requires the additional overhead associated with maintaining the same information in more than one database. It, also, makes it difficult, if not impossible, to maintain consistency and concurrency of the duplicate data. Manpower intensive solutions for eliminating inconsistencies and managing concurrency are costly and inefficient when compared to the potential of automated solutions to the problem. If an organization could identify effective and efficient automated methods for integrating existing databases it could significantly enhance the usefulness of the information it already maintains by eliminating database organizational inefficiencies which cause data duplication and inconsistencies.

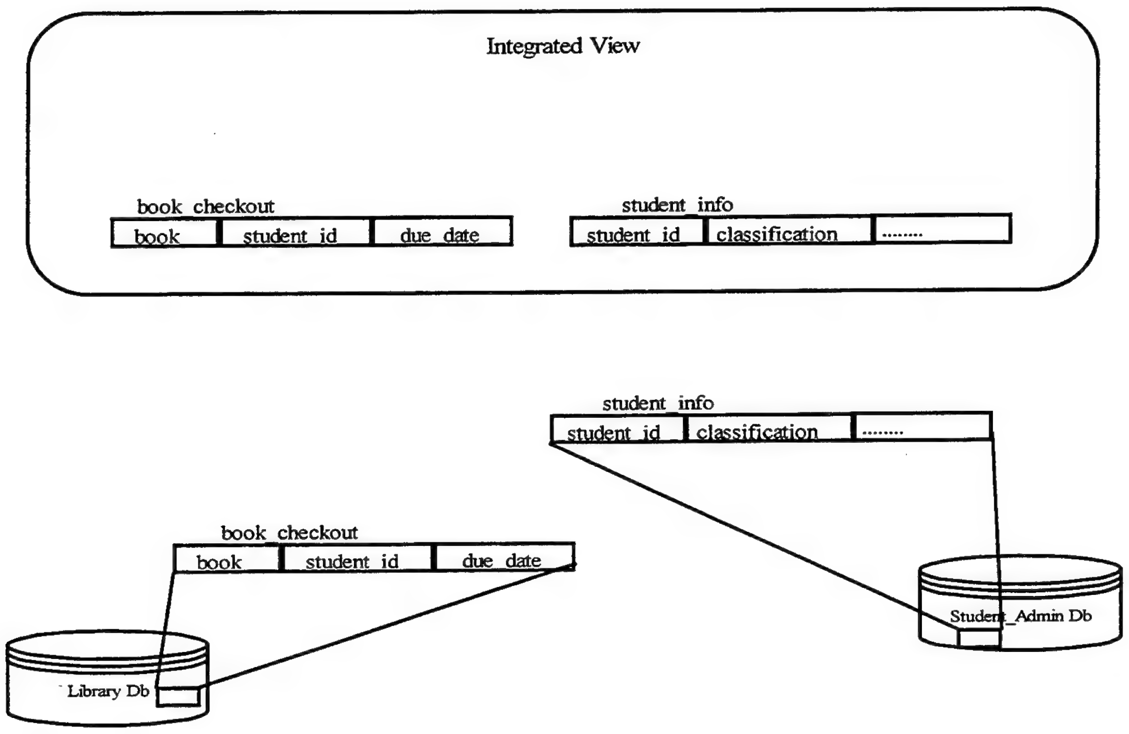


Figure 1. Integrated view of tables from disparate databases

B. THESIS OBJECTIVES

The purpose of this thesis is to determine if it is possible to concurrently access multiple independent databases within an organization in a way that provides an integrated view of the information they contain. Furthermore, assuming that this is possible, can multiple databases be efficiently and effectively integrated into a single cohesive database management system to provide a focal point of access for database applications? What are the limitations in the context of the relational model and in the context of the current database application development technology? In the process of addressing these questions this thesis also provides some analysis of the different methods which might be used to integrate multiple databases and lists the relative advantages and disadvantages of these methods.

C. METHODOLOGY

In order to test the actual ability to integrate multiple databases a relatively simple subset of the overall larger problem was selected. The problem model was built around a university academic department organization with only two specific database applications, each supported by a dedicated database. PowerBuilder 5.0, which is a relational database management system front-end development tool, was used to develop a course scheduling and management application and supporting database. This application was designed to provide a university academic department with a tool to manage and plan course scheduling and course assignments to teachers. A method was then developed to integrate the course scheduling and management database with a pre-existing academic department financial management system database which was also developed using PowerBuilder 5.0. This method was then evaluated and conclusions were developed.

D. ORGANIZATION OF THE THESIS

This thesis is organized as follows: Chapter II presents a brief discussion of the relational database model, its impact on database design, its strengths for managing and accessing data, as well as its limitations. The primary reason for this discussion is to set the frame work for any limitations which may occur in implementing a multiple database solution. Chapter III provides an overview and some discussion of the various theoretical approaches which might be taken to develop an integration solution as well as potential advantages and disadvantages. Chapter IV presents a case study of one of the approaches discussed in Chapter III and evaluates this solution against the desired goals of database integration. Chapter V looks at the various database integration performance issues in the context of the relational model and considers the impact of these issues in terms of the model's strengths as well as its limitations. Finally, Chapter VI will present conclusions drawn from this research.

II. RELATIONAL DATABASE MODEL

A. RELATIONAL MODEL OVERVIEW

The relational model has been the subject of a great deal of research since its introduction by Codd in the early 70's. In the 80's it began to replace many preexisting networking and hierarchical database systems and is currently the basis of most commercially available database management systems. Although, this may be changing as more and more organizations are looking for ways to improve their information management and access capabilities through the use of object oriented solutions such as the ODMG database model. [Ref. 1] In the relational model, data is stored as relations which are frequently referred to as tables. However, it is important to remember that these two terms are not completely interchangeable since a relation is best represented as a mathematical set of tuples which has no ordering on its members. A table however usually has some sort of order imposed on its members since it represents an actual file or some other physical implementation of an instance of a relation. A relation or table consists of rows or tuples. Because by definition, a relation is a set of tuples, each row is distinct and corresponds to an instance of an entity or a relationship which exists between two or more entity instances. The members in a row or tuple are the set of attributes of the entity or relationship instance. The attributes which correspond to the columns of a table are the properties of the entity or relationship instance. Using this basic structure, database schemas can be defined by a set of relation schemas or definitions and a set of integrity constraints placed upon those schemas. There are three fundamental update operations for a relation and they are limited by the integrity constraints on the relation schemas. They are MODIFY, UPDATE, and, DELETE. Although Codd originally defined eight relational algebra operations in the relational model for manipulation of relation instances, essentially only five of these operations are primitive. These operations are SELECT, PROJECT, UNION, DIFFERENCE, and CARTESIAN PRODUCT. [Refs. 2, 3] A more detailed discussion of relation schemas, integrity constraints, update operations, and relational algebra operations can be found in [Ref. 4]. It is this relatively simple framework which has formed the basis of a large number of relational database management systems in recent years.

B. STRENGTHS

- The relational model is based on a simple and uniform data structure and is therefore easier to understand than other database models such as the hierarchical and network data models.
- It is strongly founded on the mathematical concepts of predicate logic and set theory allowing the representation and

manipulation of data to be formalized. [Ref. 3] This is the basis of SQL, which is one of the most widely used query languages in database management systems.

- Allows a more abstract representation of a database than previous models such as the hierarchical and network data models. [Ref. 3] Therefore it is easier to design a database model which more closely resembles real world instances and relationships.

C. LIMITATIONS

- Insufficient semantic completeness or expressiveness. [Ref. 5] This has been a very popular topic for discussion and many have proposed semantic modeling extensions to the relational model to improve it. [Refs. 5, 6, 7] A good example of a semantic modeling extension is the Entity-Relational (ER) model. It is one of the most well known semantic data models because it is simple and easy to understand. However, when an ER model is converted into a database schema it may often lose its resemblance to the real world entities and relationships it was intended to represent. [Refs. 3, 4]

- Mathematical aggregate functions cannot be expressed in relational algebra. In order to specify operations such as

SUM, or AVERAGE, additional operators must be defined by the DBMS. [Ref. 4]

- The recursive closure operation cannot be specified in relational algebra. In order to specify a query to retrieve all instances in a recursive relationship, some form of looping mechanism is needed as well as a means to specify the number of recursive levels required for the specific query based on some base condition. [Ref. 4]

- OUTER JOIN and OUTER UNION operations are required to extend the relational JOIN and UNION operations to handle joins between relations or tables which either have some tuples or records which contain null values in the join attributes or have tuples or records which are not union compatible. [Ref. 4]

- The relational JOIN can be a very time consuming operation and is frequently the cause of performance degradation. This is because it requires the cross-referencing of a tuple at a time between two relations and there are many possible ways to execute this operation. As a result, complex queries are usually a performance bottleneck for large database systems. [Refs. 4, 7]

III. INTEGRATION METHODS

A. MULTIPLE DATABASE INTEGRATION

What is meant by the term multiple database integration? The term integration itself means to make into a whole by bringing all parts together or to make one thing a part of something else. This definition is applied in the context of database design in this thesis to mean the logical or physical combination of multiple databases into one database. Integration of multiple databases can be particularly useful when the same information is represented by data in multiple databases or where real world relationships exist between entities represented by data in disparate databases. In fact, within the context of a relational model, some duplication of data in disparate databases is actually necessary in order to create logical relationships which model the real world relationships not yet realized within the organization's information infrastructure. Given the above meaning for multiple database integration, data in disparate databases can then be accessed from a single vantage point. It is then possible to present new and more expanded information views derived from data spread over multiple databases and to automate more efficient procedures for eliminating inconsistencies between them. Following are a taxonomy of strategies which might provide database integration solutions as well as a brief discussion of probable advantages and disadvantages.

B. PHYSICAL INTEGRATION

The physical integration of a database is conceptually a straight forward process and consists of designing a single database which contains the union of all information contained in the preexisting databases which are to be integrated (Figure 2). The actual process of designing this new

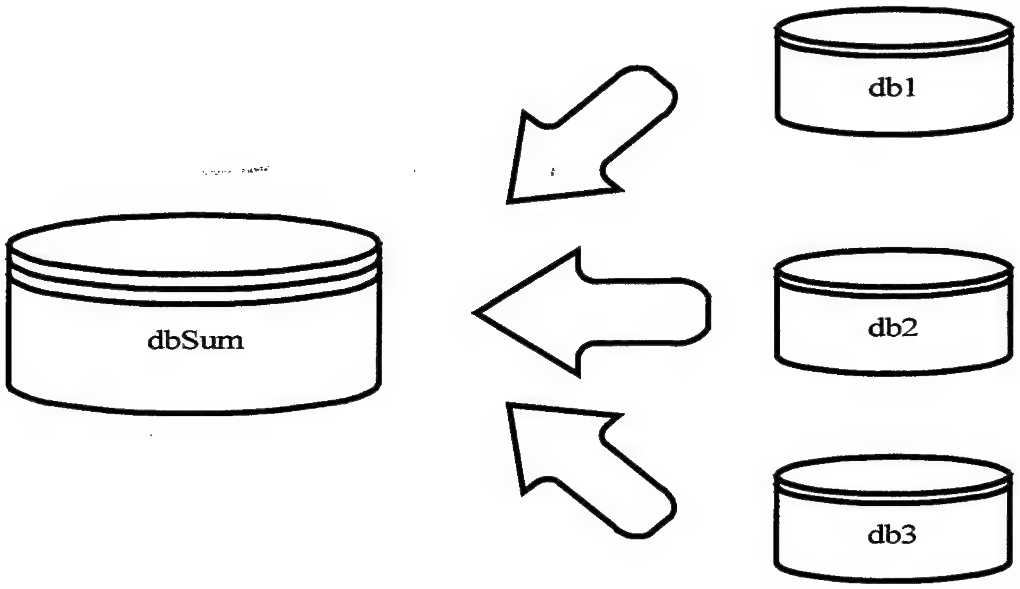


Figure 2. Physical Database Integration

database can be very difficult because it requires a complete system redesign. It will almost always require changes to any preexisting applications and, in some cases, these changes will be substantial. The most obvious benefits to this approach over other integration solutions involving a more loosely coupled arrangement are better performance and easier and less expensive application development for data access and retrieval after a physical integration solution is achieved. In a large organization with very large information management, storage, and access requirements, the physical integration will result in what is now most frequently referred to as a Very Large Database (VLDB). [Ref. 8] In this case redesign will require special attention to ensure that a number of new challenges are handled. One very large possible downside to this strategy is that it may not be as extensible as other more modular strategies which maintain the existence of separate databases. Special care is also required to achieve a solution which is scaleable in order to accommodate future expansion to handle new information requirements and avoid the need for costly changes and/or complete system redesign. Therefore, in terms of system maintenance and updates as a result of changes in the organization's business and management practices, this solution could easily become cost prohibitive. In other words, as soon as the next new requirement for information handling and access is identified, a strategy of physical integration could result in the organization being back in the same situation it was in when it started out on the path to integration.

C. LOGICAL INTEGRATION

1. Database Level

The strategy behind logical integration at the database level is the use of a middleware layer which connects to each database separately, yet presents a single interface to the application layer so that multiple databases appear to applications as one large database (Figure 3). A significant advantage to this approach is that it effectively maintains the same level of simplicity in query formulation as the physical integration strategy while at the same time allowing the use of existing databases with fewer or no changes to existing structure. This strategy has little to no effect on existing applications and only requires changes where necessary to accommodate changes in their underlying databases due to field type inconsistencies between duplicate data. The connections between preexisting databases and their applications are not affected. This strategy could be implemented through a kind of universal database engine which establishes a single connection to each database. Its job is to receive cross-database queries from an application and parse it into the appropriate subquery components for each database involved. For example, using the example of the university library given in Chapter 1, the universal engine would take the query based on the question "How many sophomores have books checked out?" and divide it into two subqueries. The subquery for the library database would be based on the smaller question "Does student A, who is a sophomore, have any books checked out?" and the subquery for the student admin database would be "Retrieve a list of all students who are sophomores." The universal engine then performs the join of the two intermediate result sets and applies an aggregate function to return the final answer to the original question. At the same time the engine should perform query optimization based on some acquired knowledge

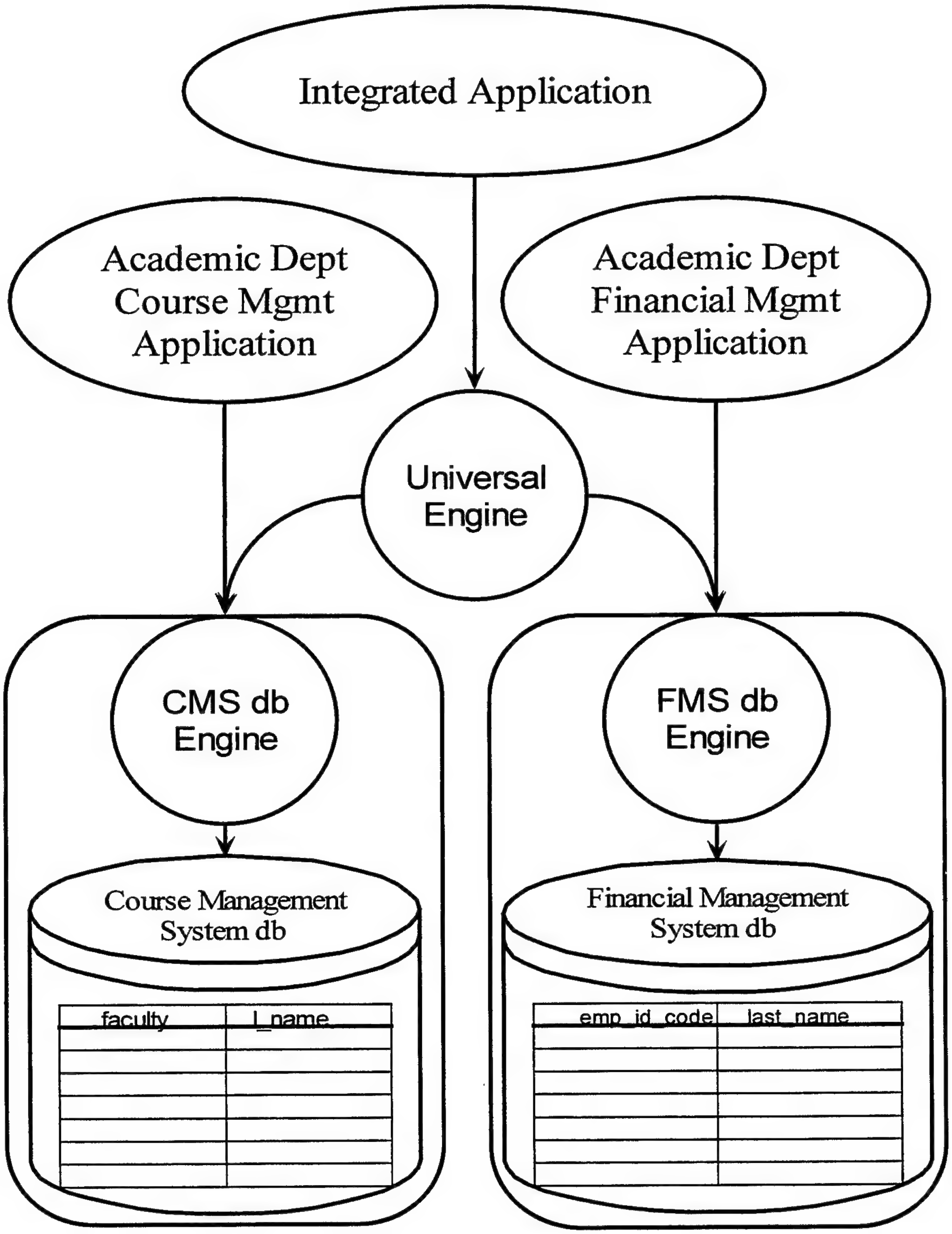

Integrated Application
Academic Dept Course Mgmt Application
Academic Dept Financial Mgmt Application
Universal Engine
CMS db Engine
FMS db Engine
Course Management System db
Financial Management System db
faculty
l_name
emp_id_code
last_name

Figure 3. Logical Integration at the Database Level

of the underlying databases structures. For example, it is probably faster to get the result above by checking the library database for the existence of a particular “student_id” after first retrieving a list of sophomores if the university has a very large number of students in comparison to the total number of books checked out. This logical integration strategy benefits largely from its more modular approach and its ability to leverage existing information infrastructure. Its modularity allows the use of databases which may be distributed and performance can benefit from a greater degree of parallelism since different parts of a cross-database query can be processed simultaneously. Additionally, this more modular and loosely coupled approach allows easier modification without necessarily affecting other parts of the organization’s information infrastructure. This can allow an organization to upgrade its systems gradually, and has more flexibility to accommodate changes as the organization’s information needs change over time. The key to the success in this strategy is the universal engine itself. It sounds nice theoretically, but it requires features that are difficult to achieve. Fortunately the first of these is already here. The open database connectivity (ODBC) standard established by Microsoft is the kind of common interface necessary for an application to connect to multiple relational database management systems. The more difficult part of the universal engine concept is the capability to establish a knowledge base from the connected databases making it possible to perform intelligent and effective query optimization. Because this solution carries with it the additional overhead of the middleware layer it might not be expected to perform as well as a more tightly coupled approach, yet its distributed architecture may actually yield better overall performance particularly in situations where there is heavy user load or when handling large queries which span multiple databases. Resolving data type mismatches where cross-database queries are involved is more of a challenge with this integration strategy as well as the question of how to deal with duplicate data which leads to concurrency and consistency problems. Maintenance of concurrency and consistency between duplicate and related data could be handled by the implementation of business rules in the middleware layer for data which is related across database boundaries. How this would be implemented and how well it would work is not clear.

2. Application Level

Integration at the application level could be accomplished by defining connections within an application to preexisting databases as illustrated in Figure 4. This approach is similar to the previous logical integration solution in that it is a more modular approach, but results in a more tightly coupled solution since the boundaries become a little more blurred. Integration of the information contained in the disparate organizational databases is handled completely at the application level. Therefore, all cross-database query optimization must be handled by the application

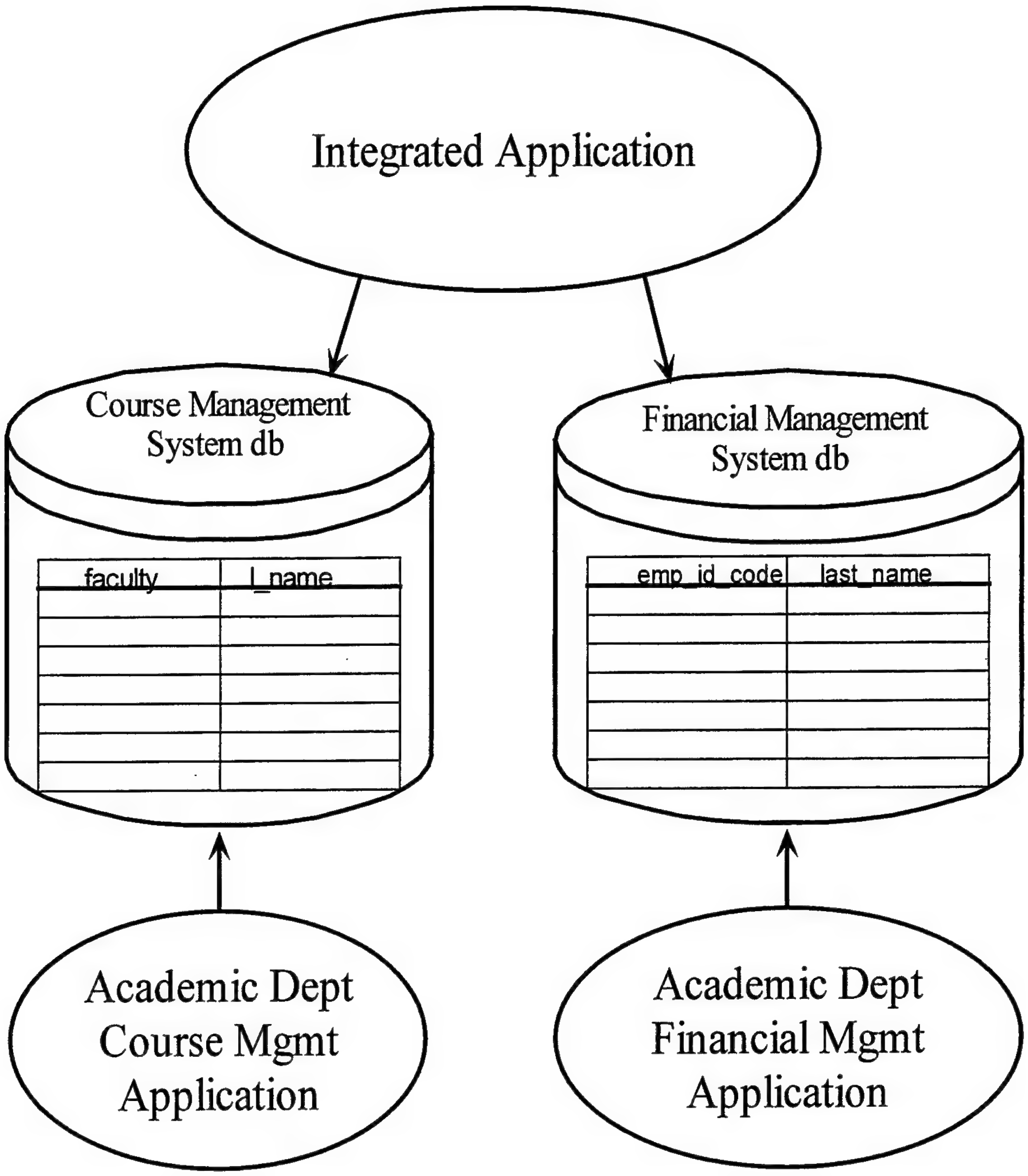

Integrated Application
Course Management System db
faculty
l_name
Financial Management System db
emp_id_code
last_name
Academic Dept Course Mgmt Application
Academic Dept Financial Mgmt Application

Figure 4. Application Level Integration

developer. This makes the application development process much more complex. The effectiveness of this approach will also depend to a great extent on the robustness of the application development environment.

Because this is a loosely coupled approach, it does not require significant redesign of the existing system, however it will require a more deliberate application design effort to resolve data conflicts, minimize duplicate data, and to maintain data concurrency and consistency between the different databases.

D. INTEGRATION STRATEGY CHALLENGES

1. Data Type Conflicts

A challenge that is common to all of the above integration strategies is how to resolve differences in data types between cross-database join fields. The differences can range from differences in field length to different types to different field masks. For example, dates in one database might be defined in "dd/mm/yy" format and in "dd/mmm/yyyy" format in another. Using a physical integration strategy, these differences are resolved in the design of the integrated database and resulting changes in data types and field lengths will most likely require changes to existing applications. However special care must be taken to avoid corruption of data during the process of migrating data from preexisting databases. Resolution of data conflicts in the logical integration strategies will most likely require changes to both existing databases and applications. However, data conflict resolution should only be necessary where actual joins across database boundaries are required. These cases can then be resolved as they are encountered during the integration design process.

2. Duplicate Data

Another common problem related to that of data conflicts is the existence of duplicate data. As we stated earlier, one of the goals of database integration is to minimize duplication between databases. This is resolved automatically through proper normalization in the physical integration strategy, but requires special attention with both logical integration approaches. In both application level and database level logical integration approaches, minimizing duplicate data will require changes to existing databases and applications. Integration at the application level will require a more substantial redesign effort but can be done as part of the actual integration implementation while at the database level fewer changes to applications should be required.

3. Concurrency and Consistency

Concurrency and consistency issues arise primarily from the existence of duplicate data and are therefore not expected to present any special problems for the physical integration strategy. Apart from concerns over wasted storage and performance inefficiencies, concurrency and consistency issues are the primary reason for minimizing duplicate data and they present significant challenges for the logical integration strategies. In a single database, consistency is handled by integrity constraints on the database schema. The problem with the logical strategies is there is no way to impose actual integrity constraints on related fields in different databases. Instead, business rules must be used to maintain consistency and must be defined in the application for application level integration if this is possible with the application development tool in use. If it is not possible, then the benefits of integration may be limited to read only reports for decision support. Business rules to maintain consistency should be defined in the universal engine for logical integration at the database level. Concurrency controls will most likely have to be implemented using procedural constraints in the application layer and may also be limited by the application development tool in use.

IV. CASE STUDY: INTEGRATION OF COURSE MANAGEMENT AND FINANCIAL MANAGEMENT DATABASES

A. BACKGROUND

1. Financial Management System

The Financial Management System (FMS) is a preexisting database application developed using PowerBuilder 5.0. Its purpose is to provide a university academic department with an automated decision support tool for managing the department's financial resources and fiscal responsibilities. The Entity-Relationship diagram for the relevant entities of this database is illustrated in Appendix A and the associated schema is contained in Appendix B. See [Ref. 9] for full details of this application and supporting database.

2. Course Management System

The Course Management System (CMS) database application manages course scheduling and course assignments to instructors and professors. It was developed using PowerBuilder 5.0. The Entity-Relationship diagram for the CMS database is illustrated in Appendix C and the associated schema is contained in Appendix D.

3. InterDatabase Relationships

In the case of the FMS and CMS databases, there is a need to accomplish some level of integration since many of the professors which are assigned to teach classes are also principle investigators (PI) for one or more research projects. This results in a natural intersection of the information contained in the two databases. The academic department would like to be able to track the relationship of courses taught by PIs since there is a business rule which links the research dollars for each PI to the number of classes he or she is required to teach. On further inspection there may be other relationships between the two databases which could be taken advantage of, but the situation described above was sufficient to answer the primary questions of this thesis. We wanted to demonstrate whether or not it was possible to first establish client/server connections to multiple databases simultaneously and, if so, perform inner joins on tables from each of those databases.

B. INTEGRATION TECHNIQUE

1. Create CMS

Design and create the CMS application and supporting database based on the need for an academic department to schedule courses and make course assignments to professors and instructors. The CMS application also serves as the integration application in order to test cross-database join techniques which can be used in logical integration at the application level.

2. Multiple Database Connections

Initially it was not clear if and how multiple simultaneous database connections could be created within an integrated database application. We discovered that this is a relatively simple process since PowerBuilder provides for the creation of multiple “transaction” objects within an application which can then be used to represent a client connection to a database on the server. The code in Figure 5 is an excerpt from the “application open” event script contained in Appendix E shows how this was done. The “cms_trCursor” and ‘fms_trCursor” are declared as gobal “Transaction” type variables in the PowerBuilder script painter. The script statement “cms_trCursor = SQLCA” assigns the “cms_trCursor” to point to the default database connection profile “SQLCA” for the CMS application. The script statement “fms_trCursor = CREATE transaction” creates a new “Transaction” object and assigns it to the “fms_trCursor”. The necessary database connection profile parameters are retrieved from the database profile sections of the “PB.ini” file using the “ProfileString()” function. The two Transaction objects are then connected to their respective databases using the “CONNECT” statement.

```
//Initial default database transaction
//In this application this transaction
//is used to connect to the CMS database
SQLCA.DBMS   = ProfileString("PB.INI","Database","DBMS"  , " ")
SQLCA.DbParm = ProfileString("PB.INI","Database","DbParm", " ")

//Transaction cms_trCursor declared in "declare:global" menu
cms_trCursor = SQLCA

CONNECT using cms_trCursor;
IF cms_trCursor.SQLCODE <> 0 THEN
  MessageBox("Connect Error", "CMS connection failed")
  HALT
END IF

//Transaction fms_trCursor declared in "declare:global" menu
fms_trCursor = CREATE transaction
fms_trCursor.DBMS   =  ProfileString("PB.INI","PROFILE Fms","DBMS"  ," ")
fms_trCursor.DbParm =  ProfileString("PB.INI","PROFILE Fms","DbParm"," ")

CONNECT using fms_trCursor;
IF fms_trCursor.SQLCODE <> 0 THEN
        MessageBox("Connect Error","FMS connection failed!")
        HALT
END IF
```

Figure 5. Application script for concurrent database connections.

3. Cross-database Join Query

The first step to developing a method or technique for performing a cross-database join is to create a standard SQL query which represents the question you wish to answer that is based on the tables in the relevant databases. The tables are identified using dot notation following the database name. The query in Figure 6 reflects the previous example relationship between the CMS and FMS databases and returns a list of faculty members who are PIs for only one research project and who are teaching less than two courses.

```
SELECT faculty_id, l_name, f_name, mi
FROM   CMS.faculty f, FMS.employee e
WHERE  f.faculty_id = e.emp_id_code
   and EXISTS (SELECT  f.faculty_id
               FROM    CMS.teach t
               WHERE   t.teach_fac_id = f.faculty_id
               GROUPBY f.faculty_id
               HAVING COUNT f.faculty_id < 2)
   and EXISTS (SELECT  e.emp_id_code
               FROM    FMS.pi p
               WHERE   p.emp_id_code = e.emp_id_code
               GROUPBY e.emp_id_code
               HAVING COUNT e.emp_id_code = 1)
```

Figure 6. Cross-database query between CMS and FMS

4. Query Subcomponents

Once the proper cross database query is identified, it is broken into subcomponent queries necessary to retrieve data from the separate databases. These subcomponents the will form the basis for the PowerBuilder "data window" objects which will have to be defined and created. The first subcomponent identified is based on data contained in the CMS database and is the part of the above query which retrieves all faculty members who are teaching less than 2 courses. (Figure 7)

```
SELECT   f.faculty_id
FROM     CMS.teach t, CMS.faculty f
WHERE    t.teach_fac_id = f.faculty_id
GROUP BY f.faculty_id
HAVING COUNT f.faculty_id < 2
```

Figure 7. First subcomponent query

The second and only other subcomponent in this example is based on data contained in the FMS database and retrieves all employees who are PIs for only one research project. (Figure 8)

```
SELECT    e.emp_id_code
FROM      FMS.pi p, FMS.employee e
WHERE     p.emp_id_code = e.emp_id_code
GROUP BY e.emp_id_code
HAVING COUNT e.emp_id_code < 2
```

Figure 8. Second subcomponent query

5. Data Window Objects

Using Powerbuilder's SQL select painter, the actual queries representing the subcomponents are created and used as the data source for separate "data window" objects. Figure 9 and Figure 10 show the resulting data source queries for the "d_teach_less_than_2" and "d_pi_less_than_2" data window objects respectively.

```
SELECT     "faculty"."faculty_id", "faculty"."l_name",
           "faculty"."f_name", "faculty"."mi"
FROM       "faculty", "teach"
WHERE     ("teach"."teach_fac_id" = "faculty"."faculty_id")
GROUP BY   "faculty"."faculty_id", "faculty"."l_name",
           "faculty"."f_name", "faculty"."mi"
HAVING    ( count("teach"."teach_fac_id") < '2' )
ORDER BY   "faculty"."faculty_id" ASC
```

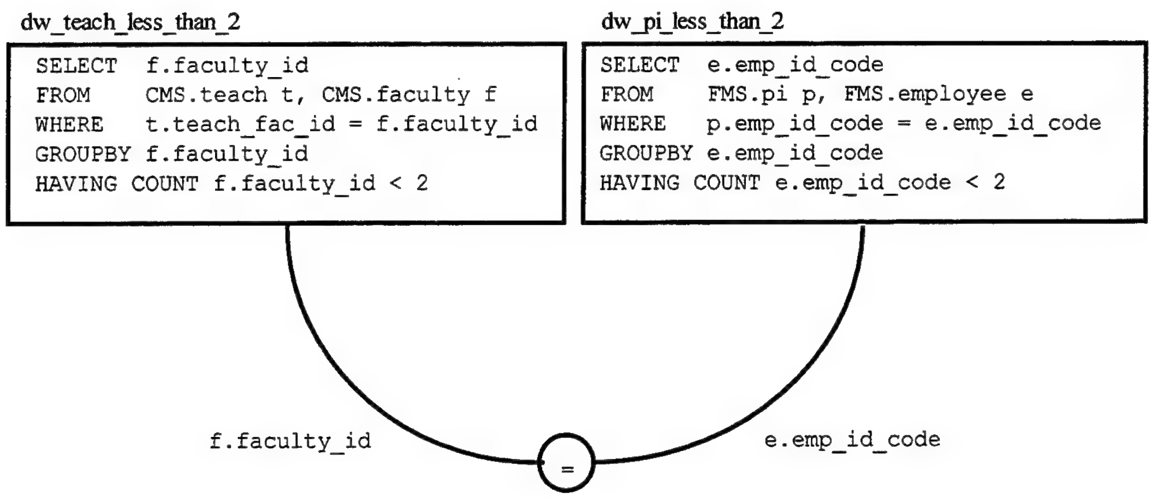
Figure 9. Data source query for "d_teach_less_than_2" data window.

```
SELECT      "employee"."emp_id_code", "employee"."first_name" ,
            "employee"."mi", "employee"."last_name"
FROM        "employee", "pi"
WHERE      ("pi"."emp_id_code" = "employee"."emp_id_code" )
GROUP BY    "employee"."emp_id_code" , "employee"."first_name",
            "employee"."mi", "employee"."last_name"
HAVING     (count("employee"."emp_id_code") = 1 )
ORDER BY    "employee"."emp_id_code" ASC
```

Figure 10. Data source query for "d_pi_less_than_2" data window.

6. Putting the Pieces Together with Windows and Scripts

Once the data windows objects are completed, they are placed in a PowerBuilder “window” using the “window painter” and code is added to the window “open” event script which implements the cross-database join between the FMS and CMS databases as reflected in the WHERE clause condition (f.faculty_id = e.emp_id_code). This approach is illustrated graphically in Figure 11. The window script to implement this approach uses nested “for loops” in order to accomplish the join cross-product of the “dw_teach_less_than_2” and “dw_pi_less_than_2” result sets. (Figure 12)



```
SELECT  f.faculty_id
FROM    CMS.teach t, CMS.faculty f
WHERE   t.teach_fac_id = f.faculty_id
GROUPBY f.faculty_id
HAVING COUNT f.faculty_id < 2
```

```
SELECT  e.emp_id_code
FROM    FMS.pi p, FMS.employee e
WHERE   p.emp_id_code = e.emp_id_code
GROUPBY e.emp_id_code
HAVING COUNT e.emp_id_code < 2
```

Figure 11. Cross-database join illustration using data windows.

```
long pi_number_of_rows
long teach_number_of_rows
long pi_loopcount
long teach_loopcount
string piNum
string teachNum

pi_number_of_rows = dw_pi_less_than_2_list.RowCount()
teach_number_of_rows = dw_fac_teach_less_than_2_list.RowCount()

for pi_loopcount = 1 to pi_number_of_rows

 piNum = dw_pi_less_than_2_list.GetItemString(pi_loopcount,1)

 for teach_loopcount = 1 to teach_number_of_rows

  teachNum =  dw_fac_teach_less_than_2_list.GetItemString(teach_loopcount,1)

  IF dw_employee_link_faculty_list.Retrieve(piNum,teachNum) >= 0 THEN
    COMMIT using SQLCA;
   ELSE
    ROLLBACK using SQLCA;
        MessageBox("Retrieve","Retrieve error - employee join faculty detail")
   END IF

  next

next
```

Figure 12. Nested for loop script to perform cross-database join

7. Performance Drag

At this point the original goal of performing a cross-database join has been accomplished with little difficulty; however, the use of nested “for loops” has a significant impact on performance. In this example, the result in the “teach less than 2” data window contains 7 records and the result in the “pi less than 2” contains 13. This results in 91 retrieval calls against the CMS.faculty table which contains 74 records. This means that 6734 join comparisons are required to get the final result which in this case yields only one name. For queries over large tables and with large intermediate result sets, this approach quickly becomes unusable.

8. A More Efficient Technique

What is needed is a more efficient method of performing the same task. If the work done in the script can be reduced to a single loop vice the nested dual loop, the performance will be significantly improved. This can be done by combining the supporting subquery for the “teach less than 2” data window with the supporting query for the “faculty_link_employee” data window and making using the resulting query the new data source for the “faculty_link_employee” data window. The result shown in Figure 13 is basically the same subquery for the condition “teach less than 2”, but we have added the join condition “faculty_id = :employee_id” to the query’s “where” clause. The string “:employee_id” is the retrieval argument passed to the data window inside a single loop in the “window” script and is as shown in Figure 14. The number of retrieval calls now required against the CMS.faculty table for the cross-database join has been reduced to 13 from 91 and the total number of join comparisons required inside the loop in the window script is reduced to 962 from 6734. When the two different windows are actually run in the application, there is a very noticeable improvement in response speed of the second technique over that of the first.

```
SELECT    "faculty"."faculty_id", "faculty"."l_name",
          "faculty"."f_name", "faculty"."mi"
FROM      "faculty", "teach"
WHERE    ("teach"."teach_fac_id" = "faculty"."faculty_id" )
    and
         ("faculty"."faculty_id" = :employee_id )
GROUP BY "faculty"."faculty_id",
          "faculty"."l_name",
          "faculty"."f_name",
          "faculty"."mi"
HAVING   ( count("teach"."teach_fac_id") < 2 )
```

Figure 13. Improved data source query for data window

```
long number_of_rows
long loopcount
string piNum

number_of_rows = dw_pi_less_than_2_list.RowCount()

FOR loopcount = 1 to number_of_rows

  piNum = dw_pi_less_than_2_list.GetItemString(loopcount,1)

  IF dw_employee_link_faculty_list.Retrieve(piNum) >= 0 THEN
    COMMIT using SQLCA;
  ELSE
    ROLLBACK using SQLCA;
    MessageBox("Retrieve", "Retrieve error - employee join faculty detail")
  END IF

NEXT
```

Figure 14. Single for loop script to perform cross-database join.

V. PERFORMANCE ISSUES

A. QUERY OPTIMIZATION

1. Relational Join Minimization

As in any relational database the join operator will have a significant impact on the performance of any integration solution. In the case of the physical integration strategy this is handled to a large extent by the query optimization rules built into the database engine. However, some attention is still required by the application developer to ensure that queries are formulated in a way that minimizes the number of intermediate result sets. In a strategy of logical integration at the database level, the implementation of query optimization rules in the universal engine is a means of minimizing this as a performance bottleneck. Integration at the application level, as we demonstrated in Chapter IV, requires special attention to query design when performing cross-database joins. As a result, large queries are probably not practical as a rule with this approach, and probably limit the usefulness of application level integration to smaller organizations and applications base on smaller databases.

2. Table Size

Although there are many ways to perform a join, one of the most common is the nested (inner-outer) loop approach, often referred to as the brute force approach. This is the method used in both techniques demonstrated in Chapter IV. We are forced to use this method in the case study because it deals with intermediate result sets or tables represented by data windows and the join is achieved through the use of a looping construct where the retrieval arguments are passed to the data window for a record by record comparison and retrieval. When using the nested loop approach, query optimization also becomes a function of which table is chosen for the outer loop and which for the inner loop. A factor known as the "join selection factor" affects the performance of a join and depends on the equijoin condition of the two tables and their size. The equijoin condition affects the percentage of records in a table which will be joined with records in the other file. In the example given in the case study, as illustrated by the "faculty" window screen capture in Appendix E, there are 13 PI employee records with less than 2 research projects retrieved from a table of 88 records in the FMS database while there only 7 faculty members teaching fewer than 2 courses retrieved from a table of 22 teach records in the CMS database. Therefore, we must perform 616 (7 x 88) join comparisons if we use the "PI less than 2" intermediate result as the outer loop versus 276 (13 x 22) if we use the "teach less than 2" intermediate result set as the outer loop. Even though the numbers in this example are relatively small, it is sufficient to show how performance can be significantly affected if one table is significantly larger than the other and only a small percentage of the records in that table are selected for join.

B. PARALLELISM

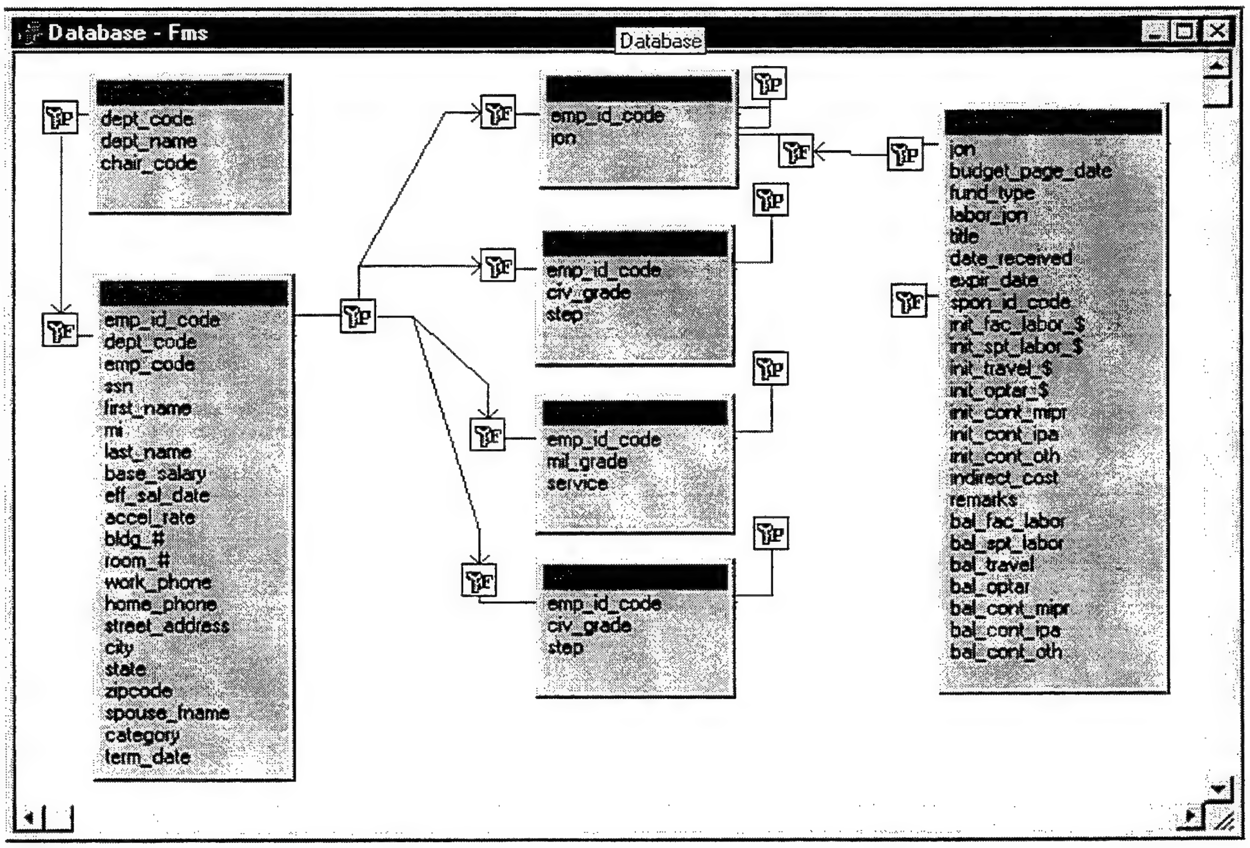
Logical integration strategies offer the ability to take advantage of parallelism which can be achieved from multiple concurrent database connections and can realize an overall performance advantage over a physical integration strategy depending on the overall system network configuration and performance. The inherently distributed nature of the logical approach avoids the problem of server overload which is more likely to occur with a single physical integrated database on a centralized server.

VI. CONCLUSIONS

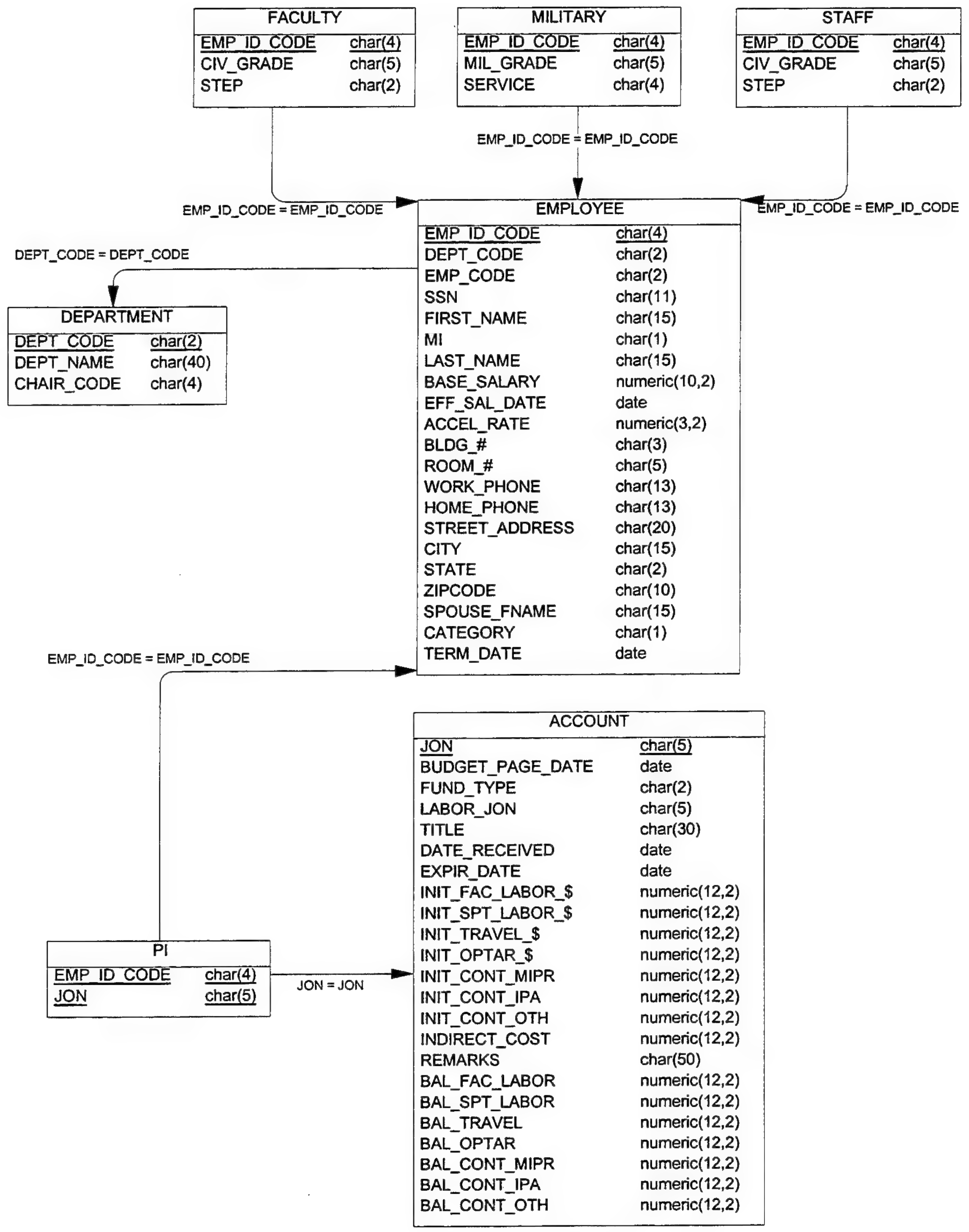
Since the introduction of the relational data model and the tremendous growth in the use of computers in recent years, the use of database applications as a tool for managing information has grown in popularity. Organizations have discovered that they now have many databases and applications which manage access to them, but they find it increasingly difficult to sift through the large amounts of information at their disposal because many of these databases contain overlapping data which is not coordinated in any useful or meaningful way. The challenge now is how to integrate existing databases so that information access and retrieval among numerous data systems is efficient and effective. We have considered three strategies for accomplishing database integration and discussed some of the probable challenges, benefits, and limitations of each in the context of the relational data model. As a result, we are reminded that one of the primary sources of difficulty in relational database retrieval is the join operation. This is even more significant when seeking an integration solution since the relational model is not conducive to cross-database join access. Using PowerBuilder 5.0 we showed techniques for performing a join across multiple database boundaries as a method of integrating existing databases at the application level. This thesis demonstrates that it is possible to perform some level of integration at the application level. However, this approach is not practical for use with large or complex queries for performance reasons making it unsuitable for use in large systems. It can be a useful means of performing limited database integration between smaller applications to leverage existing information resources to provide greater decision support utility or administrative functionality.

APPENDIX A. ENTITY RELATIONSHIP DIAGRAM FOR THE FMS DATABASE

This entity relationship diagram is a screen capture of PowerBuilder 5.0's graphic representation of the FMS database described in Chapter IV. Only tables which are relevant to this work are included.



Entity relationship diagram of FMS created using S-designor 5.0 AppModeler Desktop.



APPENDIX B. FMS DATABASE SCHEMA

EMPLOYEE

Column List

Name	Code	Type	P	M
ACCEL_RATE	ACCEL_RATE	numeric(3,2)	No	No
BASE_SALARY	BASE_SALARY	numeric(10,2)	No	No
BLDG_#	BLDG_#	char(3)	No	No
CATEGORY	CATEGORY	char(1)	No	Yes
CITY	CITY	char(15)	No	No
DEPT_CODE	DEPT_CODE	char(2)	No	No
EFF_SAL_DATE	EFF_SAL_DATE	date	No	No
EMP_CODE	EMP_CODE	char(2)	No	Yes
EMP_ID_CODE	EMP_ID_CODE	char(4)	Yes	Yes
FIRST_NAME	FIRST_NAME	char(15)	No	No
HOME_PHONE	HOME_PHONE	char(13)	No	No
LAST_NAME	LAST_NAME	char(15)	No	Yes
MI	MI	char(1)	No	No
ROOM_#	ROOM_#	char(5)	No	No
SPOUSE_FNAME	SPOUSE_FNAME	char(15)	No	No
SSN	SSN	char(11)	No	No
STATE	STATE	char(2)	No	No
STREET_ADDRESS	STREET_ADDRESS	char(20)	No	No
TERM_DATE	TERM_DATE	date	No	No
WORK_PHONE	WORK_PHONE	char(13)	No	No
ZIPCODE	ZIPCODE	char(10)	No	No

Reference to List

Reference to	Primary Key	Foreign Key
DEPARTMENT	DEPT_CODE	DEPT_CODE

Reference by List

Referenced by	Primary Key	Foreign Key
PI	EMP_ID_CODE	EMP_ID_CODE
FACULTY	EMP_ID_CODE	EMP_ID_CODE
STAFF	EMP_ID_CODE	EMP_ID_CODE
MILITARY	EMP_ID_CODE	EMP_ID_CODE

FACULTY

Column List

Name	Code	Type	P	M
CIV_GRADE	CIV_GRADE	char(5)	No	No
EMP_ID_CODE	EMP_ID_CODE	char(4)	Yes	Yes
STEP	STEP	char(2)	No	No

Reference to List

Reference to	Primary Key	Foreign Key
EMPLOYEE	EMP_ID_CODE	EMP_ID_CODE

MILITARY

Column List

Name	Code	Type	P	M
EMP_ID_CODE	EMP_ID_CODE	char(4)	Yes	Yes
MIL_GRADE	MIL_GRADE	char(5)	No	No
SERVICE	SERVICE	char(4)	No	No

Reference to List

Reference to	Primary Key	Foreign Key
EMPLOYEE	EMP_ID_CODE	EMP_ID_CODE

STAFF

Column List

Name	Code	Type	P	M
CIV_GRADE	CIV_GRADE	char(5)	No	No
EMP_ID_CODE	EMP_ID_CODE	char(4)	Yes	Yes
STEP	STEP	char(2)	No	No

Reference to List

Reference to	Primary Key	Foreign Key
EMPLOYEE	EMP_ID_CODE	EMP_ID_CODE

DEPARTMENT

Column List

Name	Code	Type	P	M
CHAIR_CODE	CHAIR_CODE	char(4)	No	No
DEPT_CODE	DEPT_CODE	char(2)	Yes	Yes
DEPT_NAME	DEPT_NAME	char(40)	No	No

Reference by List

Referenced by	Primary Key	Foreign Key
EMPLOYEE	DEPT_CODE	DEPT_CODE

PI

Column List

Name	Code	Type	P	M
EMP_ID_CODE	EMP_ID_CODE	char(4)	Yes	Yes
JON	JON	char(5)	Yes	Yes

Reference to List

Reference to	Primary Key	Foreign Key
EMPLOYEE	EMP_ID_CODE	EMP_ID_CODE
ACCOUNT	JON	JON

ACCOUNT

Column List

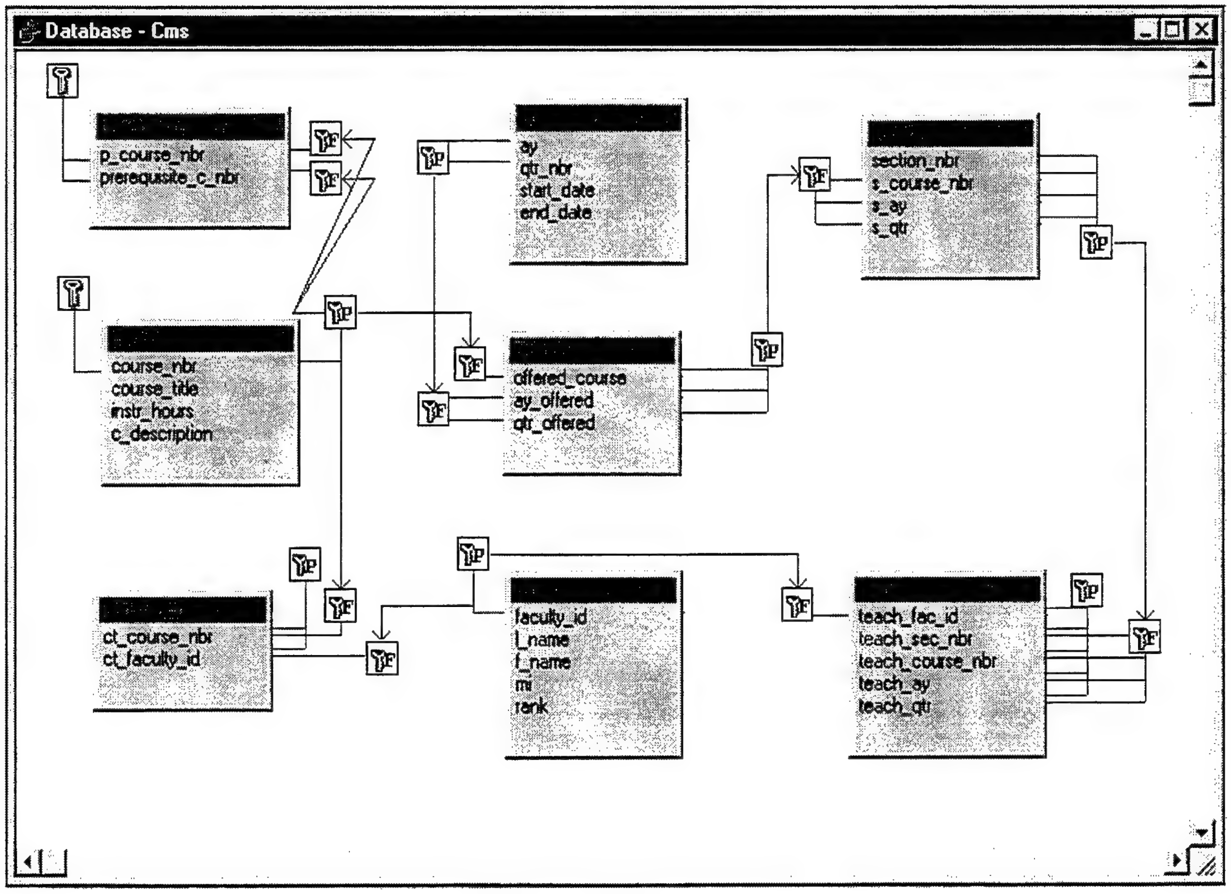
Name	Code	Type	P	M
BAL_CONT_IPA	BAL_CONT_IPA	numeric(12,2)	No	No
BAL_CONT_MIPR	BAL_CONT_MIPR	numeric(12,2)	No	No
BAL_CONT_OTH	BAL_CONT_OTH	numeric(12,2)	No	No
BAL_FAC_LABOR	BAL_FAC_LABOR	numeric(12,2)	No	No
BAL_OPTAR	BAL_OPTAR	numeric(12,2)	No	No
BAL_SPT_LABOR	BAL_SPT_LABOR	numeric(12,2)	No	No
BAL_TRAVEL	BAL_TRAVEL	numeric(12,2)	No	No
BUDGET_PAGE_DATE	BUDGET_PAGE_DATE	date	No	No
DATE_RECEIVED	DATE_RECEIVED	date	No	No
EXPIR_DATE	EXPIR_DATE	date	No	No
FUND_TYPE	FUND_TYPE	char(2)	No	Yes
INDIRECT_COST	INDIRECT_COST	numeric(12,2)	No	Yes
INIT_CONT_IPA	INIT_CONT_IPA	numeric(12,2)	No	Yes
INIT_CONT_MIPR	INIT_CONT_MIPR	numeric(12,2)	No	Yes
INIT_CONT_OTH	INIT_CONT_OTH	numeric(12,2)	No	Yes
INIT_FAC_LABOR_$	INIT_FAC_LABOR_$	numeric(12,2)	No	Yes
INIT_OPTAR_$	INIT_OPTAR_$	numeric(12,2)	No	Yes
INIT_SPT_LABOR_$	INIT_SPT_LABOR_$	numeric(12,2)	No	Yes
INIT_TRAVEL_$	INIT_TRAVEL_$	numeric(12,2)	No	Yes
JON	JON	char(5)	Yes	Yes
LABOR_JON	LABOR_JON	char(5)	No	No
REMARKS	REMARKS	char(50)	No	No
TITLE	TITLE	char(30)	No	No

Reference by List

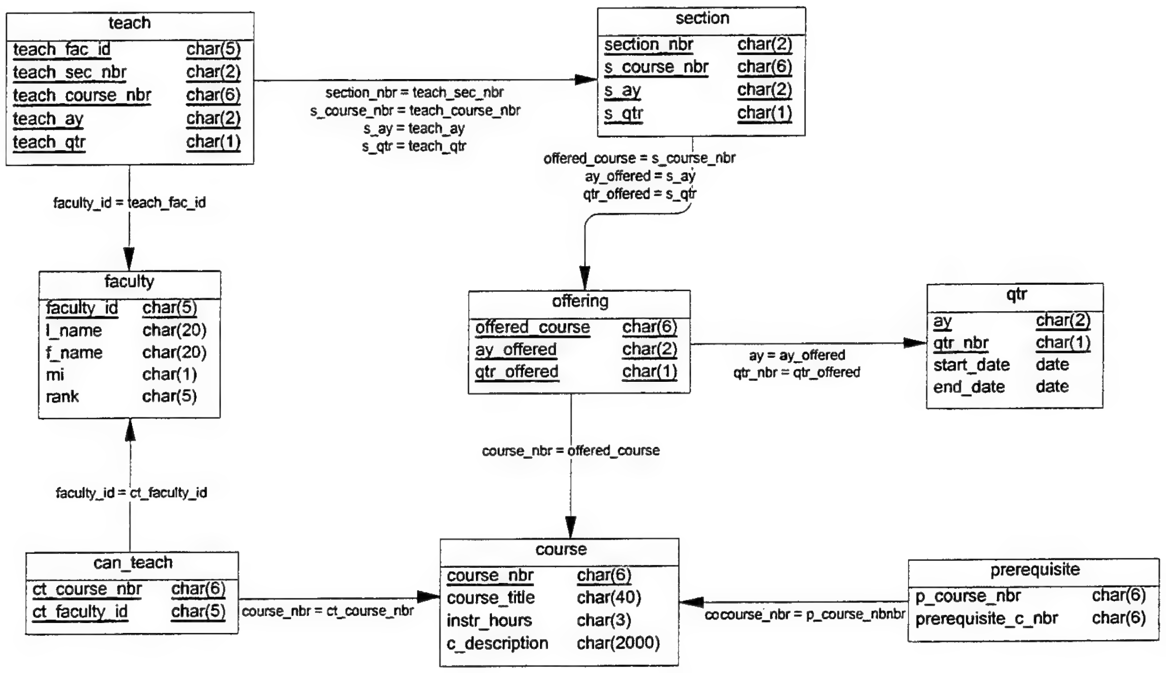
Referenced by	Primary Key	Foreign Key
PI	JON	JON

APPENDIX C. CMS ENTITY-RELATIONSHIP DIAGRAM

This entity relationship diagram is a screen capture of PowerBuilder 5.0's graphic representation of the CMS database described in Chapter IV.



. Entity relationship diagram of CMS created using S-designor 5.0 AppModeler Desktop.



APPENDIX D. CMS DATABASE SCHEMA

CAN_TEACH

Column list

Name	Code	Type	P	M
ct_course_nbr	ct_course_nbr	char(6)	Yes	Yes
ct_faculty_id	ct_faculty_id	char(5)	Yes	Yes

Reference to List

Reference to	Primary Key	Foreign Key
course	course_nbr	ct_course_nbr
faculty	faculty_id	ct_faculty_id

COURSE

Column List

Name	Code	Type	P	M
c_description	c_description	char(2000)	No	No
course_nbr	course_nbr	char(6)	Yes	Yes
course_title	course_title	char(40)	No	Yes
instr_hours	instr_hours	char(3)	No	No

Index List

Index Code	P	F	U	C	Column Code	Sort
idx_course_nbr	Yes	No	Yes	No	course_nbr	ASC

Reference by List

Referenced by	Primary Key	Foreign Key
can_teach	course_nbr	ct_course_nbr
offering	course_nbr	offered_course
prerequisite	course_nbr	p_course_nbr
prerequisite	course_nbr	prerequisite_c_nbr

FACULTY

Column List

Name	Code	Type	P	M
f_name	f_name	char(20)	No	No
faculty_id	faculty_id	char(5)	Yes	Yes
l_name	l_name	char(20)	No	No
mi	mi	char(1)	No	No
rank	rank	char(5)	No	No

Reference by List

Referenced by	Primary Key	Foreign Key
can_teach	faculty_id	ct_faculty_id
teach	faculty_id	teach_fac_id

OFFERING

Column List

Name	Code	Type	P	M
ay_offered	ay_offered	char(2)	Yes	Yes
offered_course	offered_course	char(6)	Yes	Yes
qtr_offered	qtr_offered	char(1)	Yes	Yes

Reference to List

Reference to	Primary Key	Foreign Key
course	course_nbr	offered_course
qtr	ay	ay_offered
	qtr_nbr	qtr_offered

Reference by List

Referenced by	Primary Key	Foreign Key
section	offered_course	s_course_nbr
	ay_offered	s_ay
	qtr_offered	s_qtr

PREREQUISITE

Column List

Name	Code	Type	P	M
p_course_nbr	p_course_nbr	char(6)	No	Yes
prerequisite_c_nbr	prerequisite_c_nbr	char(6)	No	Yes

Table prerequisite Index List

Index Code	P	F	U	C	Column Code	Sort
pk_prereq	No	Yes	Yes	No	p_course_nbr	ASC
					prerequisite_c_nbr	ASC

Reference to List

Reference to	Primary Key	Foreign Key
course	course_nbr	p_course_nbr
course	course_nbr	prerequisite_c_nbr

QTR

Column List

Name	Code	Type	P	M
ay	ay	char(2)	Yes	Yes
end_date	end_date	date	No	Yes
qtr_nbr	qtr_nbr	char(1)	Yes	Yes
start_date	start_date	date	No	Yes

Reference by List

Referenced by	Primary Key	Foreign Key
offering	ay	ay_offered
	qtr_nbr	qtr_offered

SECTION

Column List

Name	Code	Type	P	M
s_ay	s_ay	char(2)	Yes	Yes
s_course_nbr	s_course_nbr	char(6)	Yes	Yes
s_qtr	s_qtr	char(1)	Yes	Yes
section_nbr	section_nbr	char(2)	Yes	Yes

Reference to List

Reference to	Primary Key	Foreign Key
offering	offered_course	s_course_nbr
	ay_offered	s_ay
	qtr_offered	s_qtr

Reference by List

Referenced by	Primary Key	Foreign Key
teach	section_nbr	teach_sec_nbr
	s_course_nbr	teach_course_nbr
	s_ay	teach_ay
	s_qtr	teach_qtr

TEACH

Column List

Name	Code	Type	P	M
teach_ay	teach_ay	char(2)	Yes	Yes
teach_course_nbr	teach_course_nbr	char(6)	Yes	Yes
teach_fac_id	teach_fac_id	char(5)	Yes	Yes
teach_qtr	teach_qtr	char(1)	Yes	Yes
teach_sec_nbr	teach_sec_nbr	char(2)	Yes	Yes

Reference to List

Reference to	Primary Key	Foreign Key
faculty	faculty_id	teach_fac_id
section	section_nbr	teach_sec_nbr
	s_course_nbr	teach_course_nbr
	s_ay	teach_ay
	s_qtr	teach_qtr

APPENDIX E. CMS SCRIPTS AND SAMPLE SCREEN CAPTURES

Script for Application Open Event

```
//Initial default database transaction
//In this application this transaction
//is used to connect to the CMS database
SQLCA.DBMS    = ProfileString("PB.INI","Database","DBMS"   , " ")
SQLCA.DbParm = ProfileString("PB.INI","Database","DbParm",  " ")

//Transaction cms_trCursor declared in "declare:global" menu
cms_trCursor = SQLCA

CONNECT using cms_trCursor;
IF cms_trCursor.SQLCODE <> 0 THEN
   MessageBox("Connect Error", "CMS connection failed")
   HALT
END IF


//Transaction fms_trCursor declared in "declare:global" menu
fms_trCursor = CREATE transaction
fms_trCursor.DBMS =   ProfileString("PB.INI","PROFILE Fms","DBMS",  " ")
fms_trCursor.DbParm = ProfileString("PB.INI","PROFILE Fms","DbParm"," ")

CONNECT using fms_trCursor;
IF fms_trCursor.SQLCODE <> 0 THEN
      MessageBox("Connect Error","FMS connection failed!")
      HALT
END IF


/* let user control the toolbar*/
toolbarusercontrol = TRUE
toolbartext = TRUE

open(w_main)
```

Script for Application Close Event

```
transaction cms_trCursor
cms_trCursor = SQLCA

DISCONNECT using cms_trCursor;
IF cms_trCursor.SQLCODE <> 0 THEN
   ROLLBACK using cms_trCursor;
   MessageBox("Disconnect", cms_trCursor.SQLERRTEXT)
END IF
```

Script for w_faculty "window" Open Event with nested loop

```
// associate each data window with a database connection
// transaction object
dw_fac_teach_less_than_2_list.settransobject ( sqlca )
dw_pi_less_than_2_list.settransobject(fms_trcursor)
dw_employee_link_faculty_list.settransobject ( sqlca )

// disable edit control for each data window
dw_pi_less_than_2_list.Modify("datawindow.readOnly = yes")
dw_fac_teach_less_than_2_list.Modify("datawindow.readOnly = yes")
dw_employee_link_faculty_list.Modify("datawindow.readOnly = yes")

// retrieve list of faculty from CMS for which teach instances
// are less than two dw_fac_teach_less_than_2_list data window
IF dw_fac_teach_less_than_2_list.Retrieve() = -1 THEN
   ROLLBACK using SQLCA;
   MessageBox("Retrieve", "Faculty Teach Retrieval Error")
ELSE
   COMMIT using SQLCA;
END IF

// retrieve list of employees from FMS for which PI instances
// are less than 2 in dw_pi_less_than_2_list data window
IF dw_pi_less_than_2_list.Retrieve() = -1 then
      rollback using fms_trCursor;
      messagebox("Error","PI Employee List Retrieval Error")
ELSE
      commit using fms_trCursor;
END IF
```

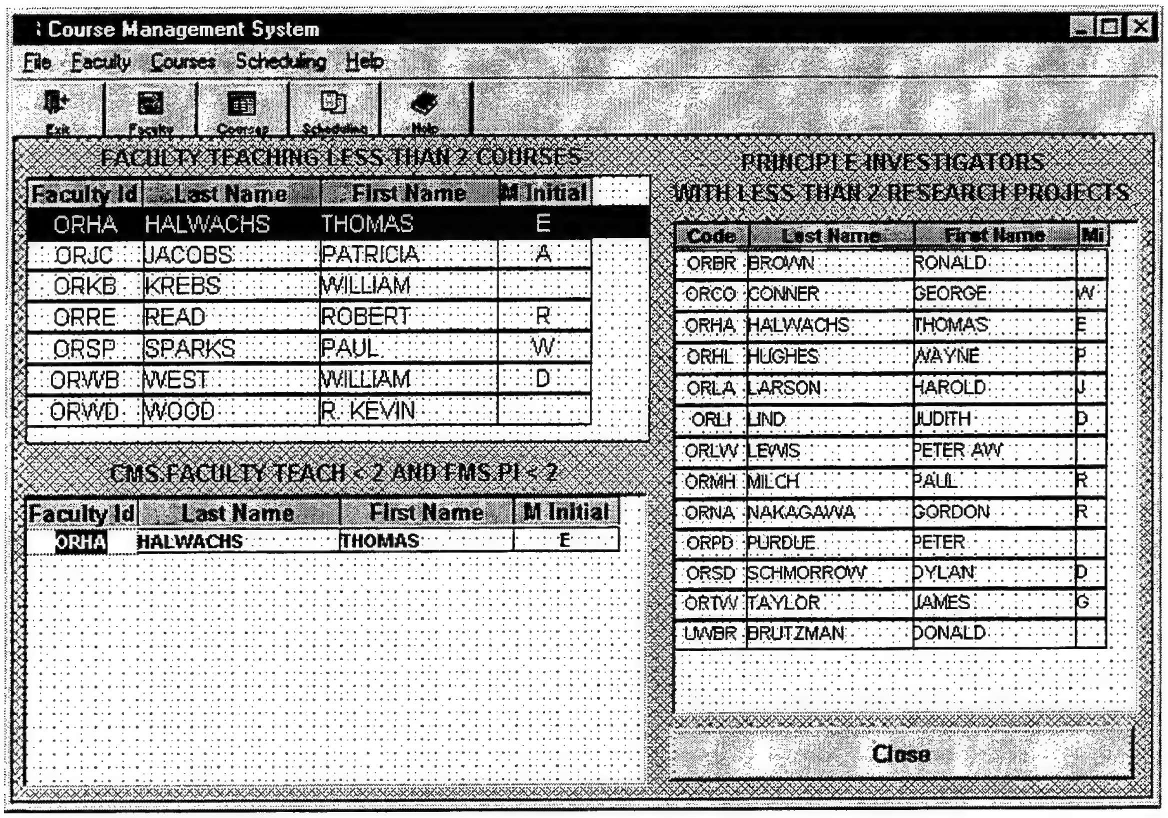
Script for w_faculty "window" Open Event with nested loop (continued)

```
long pi_number_of_rows
long teach_number_of_rows
long pi_loopcount
long teach_loopcount
string piNum
string teachNum

// get number of rows in pi result data window
pi_number_of_rows = dw_pi_less_than_2_list.RowCount()
// get number of rows in teach result data window
teach_number_of_rows = dw_fac_teach_less_than_2_list.RowCount()

for pi_loopcount = 1 to pi_number_of_rows

  piNum = dw_pi_less_than_2_list.GetItemString(pi_loopcount,1)

  for teach_loopcount = 1 to teach_number_of_rows

    teachNum =
    dw_fac_teach_less_than_2_list.GetItemString(teach_loopcount,1)

IF dw_employee_link_faculty_list.Retrieve(piNum,teachNum) >= 0
THEN
      COMMIT using SQLCA;
    ELSE
      ROLLBACK using SQLCA;
       MessageBox("Retrieve", "Retrieve error - employee join
                                               faculty detail")
    END IF

  next
 next
```

Script for w_faculty "window" Open Event with single for loop

```
// associate each data window with a database connection
// transaction object
dw_fac_teach_less_than_2_list.settransobject ( sqlca )
dw_pi_less_than_2_list.settransobject(fms_trcursor)
dw_employee_link_faculty_list.settransobject ( sqlca )

// disable edit control for each data window
dw_fac_teach_less_than_2_list.Modify("datawindow.readOnly = yes")
dw_pi_less_than_2_list.Modify("datawindow.readOnly = yes")
dw_employee_link_faculty_list.Modify("datawindow.readOnly = yes")

// retrieve list of faculty from CMS for which teach instances
// are less than two dw_fac_teach_less_than_2_list data window
IF dw_fac_teach_less_than_2_list.Retrieve() = -1 THEN
   ROLLBACK using SQLCA;
   MessageBox("Retrieve", "Faculty Teach Retrieval Error")
ELSE
   COMMIT using SQLCA;
END IF

// retrieve list of employees from FMS for which PI instances
// are less than 2 in dw_pi_less_than_2_list data window
IF dw_pi_less_than_2_list.Retrieve() = -1 THEN
  ROLLBACK using fms_trCursor;
    messagebox("Error","Employee List Retrieval Error")
ELSE
  COMMIT using fms_trCursor;
END IF


long number_of_rows
long loopcount
string piNum

number_of_rows = dw_pi_less_than_2_list.RowCount()

FOR loopcount = 1 to number_of_rows

   piNum = dw_pi_less_than_2_list.GetItemString(loopcount,1)

   IF dw_employee_link_faculty_list.Retrieve(piNum) >= 0 THEN
      COMMIT using SQLCA;
   ELSE
     ROLLBACK using SQLCA;
     MessageBox("Retrieve",
                "Retrieve error - employee join faculty detail")
   END IF

NEXT
```

Screen capture of w_faculty window



FACULTY TEACHING LESS THAN 2 COURSES

Faculty Id	Last Name	First Name	M Initial
ORHA	HALWACHS	THOMAS	E
ORJC	JACOBS	PATRICIA	A
ORKB	KREBS	WILLIAM	
ORRE	READ	ROBERT	R
ORSP	SPARKS	PAUL	W
ORWB	WEST	WILLIAM	D
ORWD	WOOD	R. KEVIN	

CMS.FACULTY TEACH < 2 AND FMS.PI < 2

Faculty Id	Last Name	First Name	M Initial
ORHA	HALWACHS	THOMAS	E

PRINCIPLE INVESTIGATORS
WITH LESS THAN 2 RESEARCH PROJECTS

Code	Last Name	First Name	Mi
ORBR	BROWN	RONALD	
ORCO	CONNER	GEORGE	W
ORHA	HALWACHS	THOMAS	E
ORHL	HUGHES	WAYNE	P
ORLA	LARSON	HAROLD	J
ORLI	LIND	JUDITH	D
ORLW	LEWIS	PETER AW	
ORMH	MILCH	PAUL	R
ORNA	NAKAGAWA	GORDON	R
ORPD	PURDUE	PETER	
ORSD	SCHMORROW	DYLAN	D
ORTW	TAYLOR	JAMES	G
UWBR	BRUTZMAN	DONALD	

LIST OF REFERENCES

1. Loomis, Mary E. S. , *Object Databases The Essentials, Reading*, MA: Addison-Wesley Publishing Company, 1996.

2. Date, C. J., *An Introduction to Database Systems*, Fourth Edition, Volume 1, Addison-Wesley Publishing Company, Reading, MA, 1986.

3. Spear, R. L., "*A Relational/Object-Oriented Database Management System: R/OODBMS,*" Master's Thesis, Naval Postgraduate School, Monterey, CA, September, 1992.

4. Elmasri, R. and Navathe, S. B., *Fundamentals of Database Systems*, Second Edition, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

5. Kent, W. Limitations of Record-Based Information Models, *ACM Transactions on Database Systems*, 4(1); 1979; 107-131.

6. Codd, E. J., Extending the Database Relational Model to Capture More Meaning, ACM Transactions on Database Systems, 4(4); 1979.

7. Wu, C. T., An Effect of Set Type to Query Formulation in Relational Database Systems, Naval Postgraduate School, Monterey, CA, Report No NPS52-88-018, July, 1988.

8. Jernigan, Kevin, "Making the Move to VLDBs," *Databased Advisor*, p. 36, March, 1997.

9. Pires, S. Alan, FMS Thesis reference, Master's Thesis, Naval Postgraduate School, Monterey, CA, March, 1997.

BIBLIOGRAPHY

Casanova, M. A., Furtado, A. L., and Tucherman, L., A Software Tool for Modular Design, *ACM Transactions on Database Systems*, Vol. 16, No.2, June 1991, pp. 209-234.

Codd, E. F., *The Relational Model for Database Management, Version 2*, Reading, MA: Addison-Wesley Publishing Company, 1990.

Date, C. J., with Warden, A., *Relational Database Writings, 1885-1989*, Reading, MA: Addison-Wesley Publishing Company, 1990.

Date, C. J., with Darwin, H., *Relational Database Writings, 1889-1991*, Reading, MA: Addison-Wesley Publishing Company, 1992.

Grahne, G., The Problem of Incomplete Information in Relational Databases, *Lecture Notes in Computer Science, Vol 554*, Springer-Verlag Berlin Heidelberg, 1991.

Kuper, G. M. and Moshe, Y. V., The Logical Data Model, *ACM Transactions on Database Systems*, Vol. 15, No.1, September 1990, pp. 379-413.

Langerak, R., View Updates in Relational Databases with an Independent Scheme, *ACM Transactions on Database Systems*, Vol. 15, No.1, March 1990, pp. 40-66.

Shasha, D. and Wang, T., Optimizing, Equijoin Queries in Distributed Databases Where Relations Are Hash Partitioned, *ACM Transactions on Database Systems*, Vol. 16, No.2, June 1991, pp. 279-308.

Markowitz, V. M. and Shoshani, A., Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach, *ACM Transactions on Database Systems*, Vol. 17, No.3, September 1992, pp. 423-464.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.. 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, VA 22134-5027

4. Director, Marine Corps Research Center 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

5. Director, Studies and Analysis Division 1
   MCCDC, Code C45
   300 Russell Road
   Quantico, VA 22134-5130

6. Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5101

7. C. Thomas Wu, Code CS/Wq 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5101

8. Mark H. Barber, Code CS/Bm 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5101

9. Maj. Barron D. Whitaker 8
375C Bergin Dr.
Monterey, CA, 93940